Learning Heuristics to solve Travelling Salesmen problem

Tejas Pagare, Kaustubh Ponkshe

CS769 Paper Presentation

Electrical Engineering Department
Indian Institute of Technology Bombay

May 6, 2022

Outline

Combinatorial Optimization

- Combinatorial optimization is an optimization problem wit discrete objects and the objective of the algorithm is to minimize or maximize a cost function.
- The problem is to find an optimal solution from a feasible set of discrete finite solutions.
- Some examples of combinatorial optimization include Travelling salesman-problem, minimum graph cut, set cover problem, etc

Applications

Travelling Salesman Problem

Definition (Travelling Salesman Problem)

"Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph exactly once"

Consider a graph $G = (V, E)$ where $V = (v_1, v_2, ...v_n)$ is a set of n vertices and there is a cost function c_{ij} associated with each edge $e_{ij} \in E$. Now consider variables x_{ij} such that .

$$x_{ij} = \begin{cases} 1 \text{ if there is a path from i to j} \\ 0 \text{ otheriwse} \end{cases}$$

Then our optimization problem can be given as

$$\min \sum_{i=1}^{n} \sum_{i \neq j, j=1}^{n} x_{ij} c_{ij}$$

with the constraint that

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \text{ for every } j \text{ and } \sum_{i \neq j, j=1}^{n} x_{ij} = 1 \text{ for every } i$$

Greedy Methods

Greedy algorithms are a class of algorithms which look for best in the short run, whether or not it is best in the long run.Greedy algorithms optimize locally, but not necessarily globally.

Definition (Nearest Neighbour Algorithm)

"The Nearest-Neighbor Algorithm begins at any vertex and follows the edge of least weight from that vertex. At every subsequent vertex, it follows the edge of least weight that leads to a city not yet visited, until it returns to the starting point."

Heursitic Based Methods

A heuristic function, also simply called a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.

Algorithm Nearest Insertion Algorithm

```
**Select** $v_1 \in V$, Intitialize subgraph $S = \phi$
Find vertex $v_j = \text{argmin}_j c_{1j}$
$S = v_1, v_j, v_1$
**for** *t=1 to n-2* **do**
    Find vertex $v_k = \text{argmin}_{v_i \in S} c_{ik}$
    Find arc (i,j) $= \text{argmin}_{v_i \in S} c_{ik} + c_{jk} - c_{ij}$
    Update$S = v_1, v_{k1}, ...., v_j, v_k, v_i \dots$
**end**
```

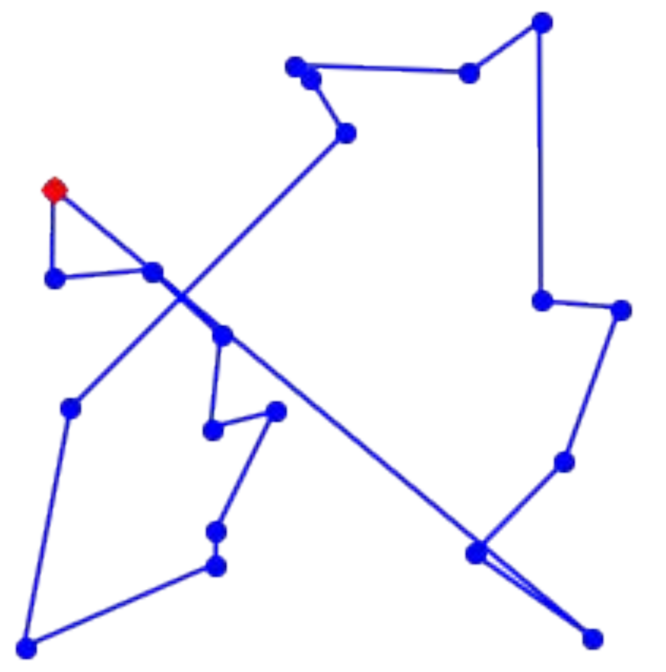
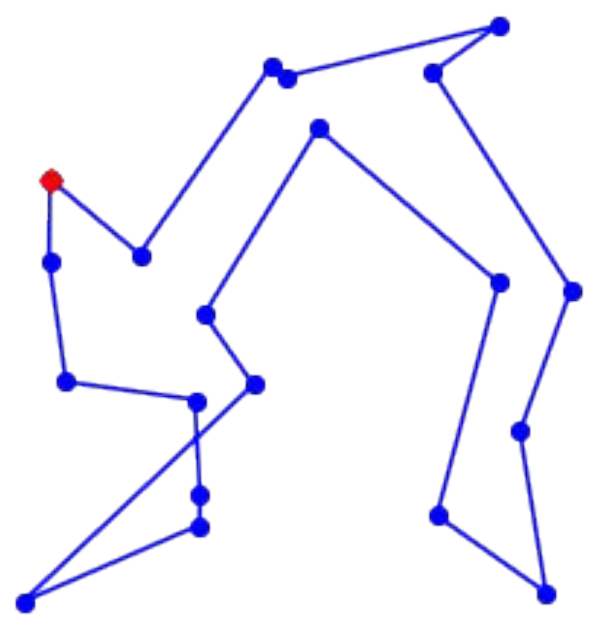
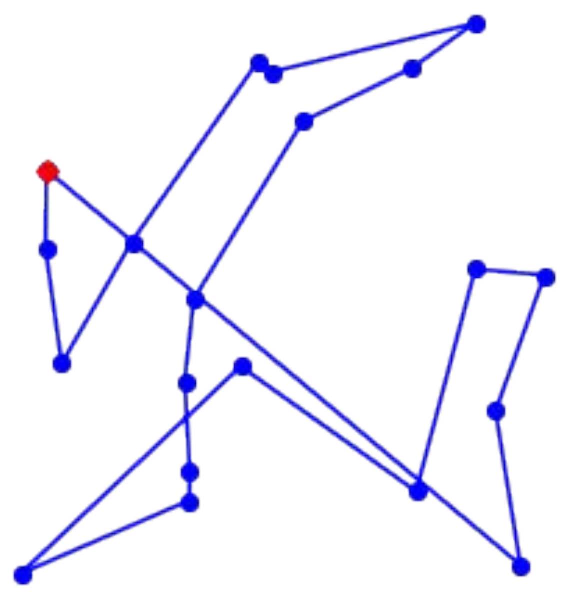
Farthest Insertion Algorithm inserts farthest points

Average and Worst case results

- **Nearest Neighbour:** Average length = 1.26*length of optimal tour, no worst case bound
- **Nearest Insertion:** worst case length = 2*length of optimal tour, $O(n^2)$ complexity
- **Farthest Insertion:** worst case length = 2*log(n)*length of optimal tour, $O(n^2)$ complexity

Experimental Results

Nodes	Nearest Neighbour	Nearest Insertion	Farthest Insertion
20	5.106	5.248	5.246
50	7.293	7.378	8.076

Learning Based Methods

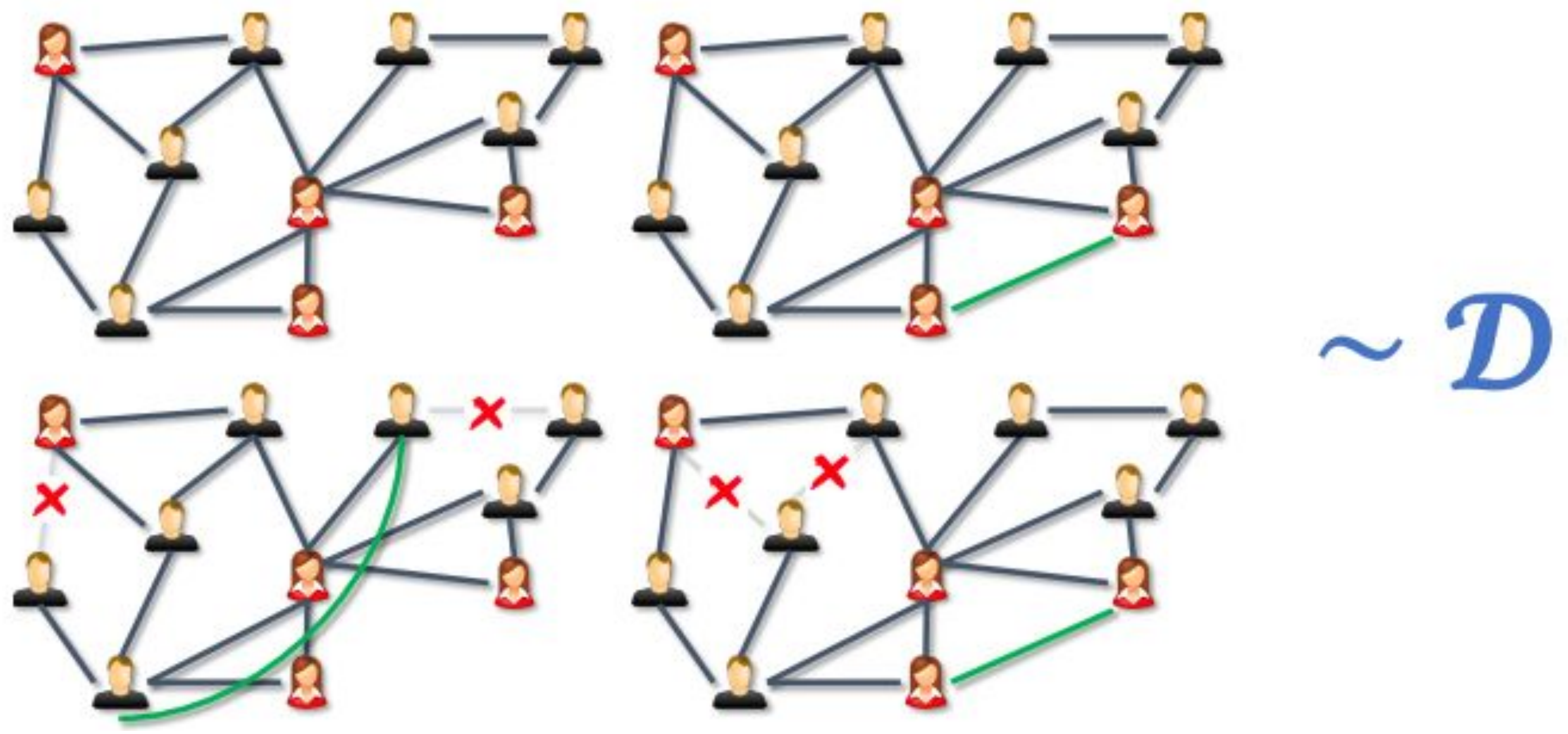
Lot of prior algorithms were Heuristic based.

Can we learn such heuristic algorithms?

Yes, using Machine Learning!

Problem Statement:

Given a graph optimization problem G and a distribution D of problem instances, can we learn better greedy heuristics that generalize to unseen instances from D?

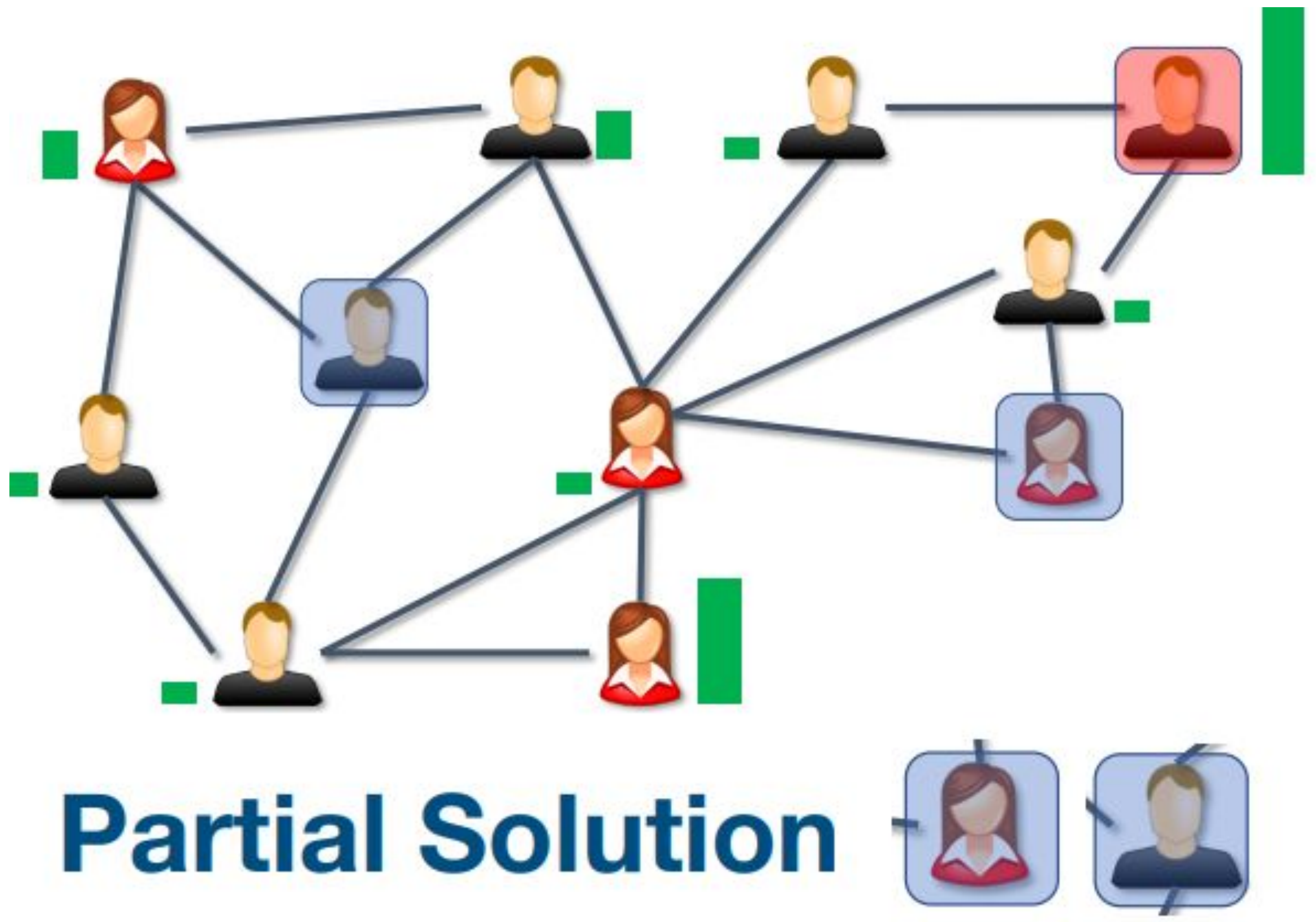


Reinforcement Learning

Greedy Algorithm	Reinforcement Learning
Partial solution	State
Scoring function	Q-function
Select best node	Greedy Policy

Algorithm:
Repeat until all edges are covered:

1. Compute node scores
2. Select best node w.r.t. score
3. Add best node to partial solution.



Markov Decision Process

Definition (Markov Decision Process)

A MDP consists of

- A set of states $\mathcal{S}$, set of actions $\mathcal{A}$ for moving from a state to another. $\mathcal{S}$ and $\mathcal{A}$ can be both finite or infinite.
- Transition Probabilities P: probability distribution over next states given the current state and current action where $P_{ij}(a) = \Pr\{X_{n+1} = j | X_n = i, U_n = a\}$.
- A reward function: $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\mathcal{R}_s^a$ or $r(s, a)$ is the expected reward of taking action a in state s.

Therefore a MDP is simply given as a pair $(\mathcal{S}, \mathcal{A}, P, \mathcal{R})$.

Reinforcement Learning (contd.)

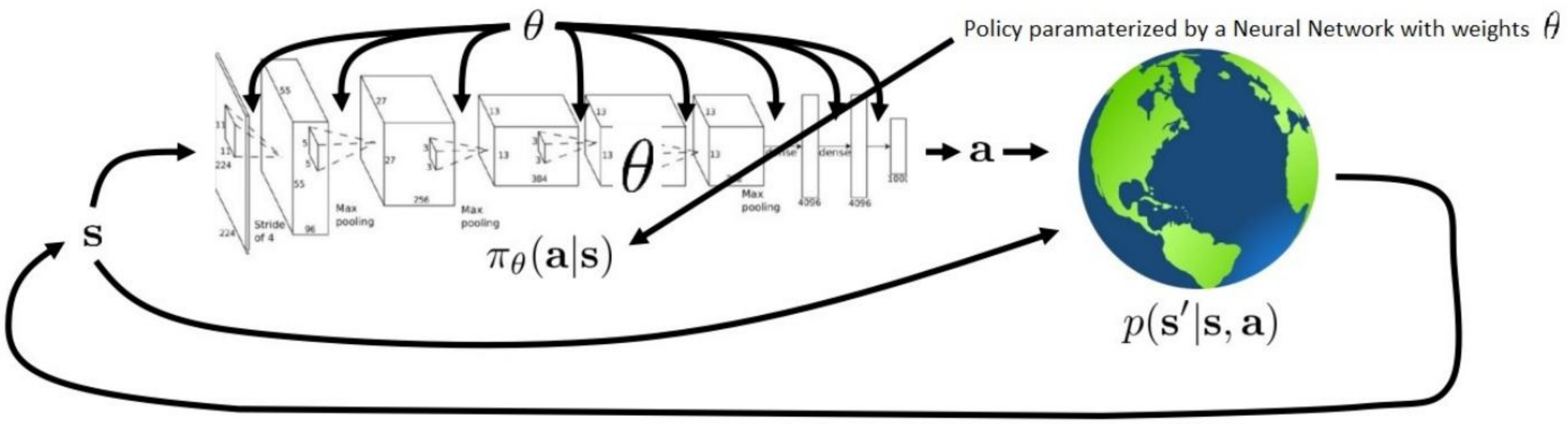

θ
Policy paramaterized by a Neural Network with weights θ
θ
$\mathbf{a}$
$\mathbf{s}$
$\pi_\theta(\mathbf{a}|\mathbf{s})$
$p(\mathbf{s}'|\mathbf{s},\mathbf{a})$
Stride of 4
Max pooling
Max pooling
Max pooling
dense
dense

Figure: Environment-Agent Interaction

We usually consider learning in a Markov Decision Process $(\mathcal{S}, \mathcal{A}, P, \mathcal{R})$ where the aim is to find

$$\theta^\star = \underset{\theta}{\text{argmax}} \mathbb{E}_{\tau \sim p_\theta(\tau)} \Big[\sum_t r(s_t, a_t) \Big]$$

where

$$p_\theta(s_1, a_t, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

REINFORCE Algorithm

Theorem (Policy Gradient Theorem)

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \Big(Q_\pi(s,a) - b(s) \Big) \nabla \pi_\theta(a|s) \tag{1}$$

where $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \Big[\sum_t r(s_t, a_t) \Big]$ *and*

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \Big]$$

is the Q-value function for policy π *and* $\mu(s)$ *is a state distribution satisfying* $\mu(s) \geq 0 \ \forall \ s$ *and* $\sum_s \mu(s) = 1$.

The policy-gradient methods seek to maximize J as follows:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$$

where $\widehat{\nabla J(\theta_t)}$ is the stochastic estimate of the actual gradient of J w.r.t θ.

REINFORCE Algorithm (contd.)

Algorithm REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi_\theta(a|s)$, a differentiable state-value function
parameterization $\hat{V}(s, w)$
Algorithm parameters: step sizes $\alpha_1 > 0, \alpha_2 > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value parameters $w \in \mathbb{R}^d$
for each episode do
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi_\theta(\cdot|\cdot)$
    for step t=0,1,...,T-1 do
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \hat{V}(s, w)$
        $w \leftarrow w + \alpha_1 \delta \nabla \hat{V}(S_t, w)$
        $\theta \leftarrow \theta + \alpha_2 \gamma^t \delta \nabla \ln \pi_\theta(A_t|S_t)$
    end
end
```

Attention, Learn to Solve Routing Problems!

Consider for the n-node graph problem instance s, the solution tour $\pi = (\pi_1, \dots, \pi_n)$ as the permutation of nodes, $\pi_t \in \{1, \dots, n\}$. The aim is to find a stochastic policy factorized using $p_\theta(\pi|s) = \prod_{t=1}^{n} p_\theta(\pi_t|s, \pi_{1:t-1})$.

Here, we consider the an encoder-decoder architecture similar to Transformer (Vaswani et al., 2017).

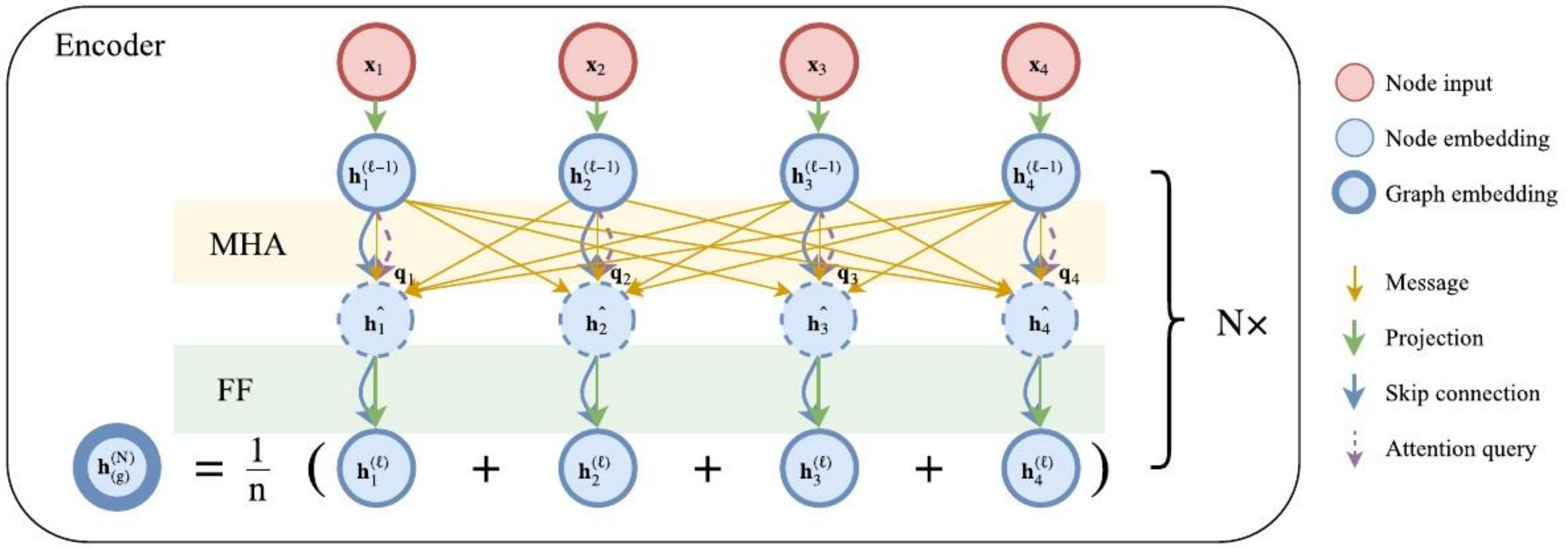


Figure: Attention based encoder

x_i: d_x dimensional input feature for node i
$h_i^{(0)}$: learned d_h dimensional node embedding
Output: node embedding for each node i, $h_i^{(N)}$ and graph embedding $\bar{h}^{(N)} = \frac{1}{n}\sum_{i=1}^{n} h_i^{(N)}$

Attention Based Algorithm (contd.)

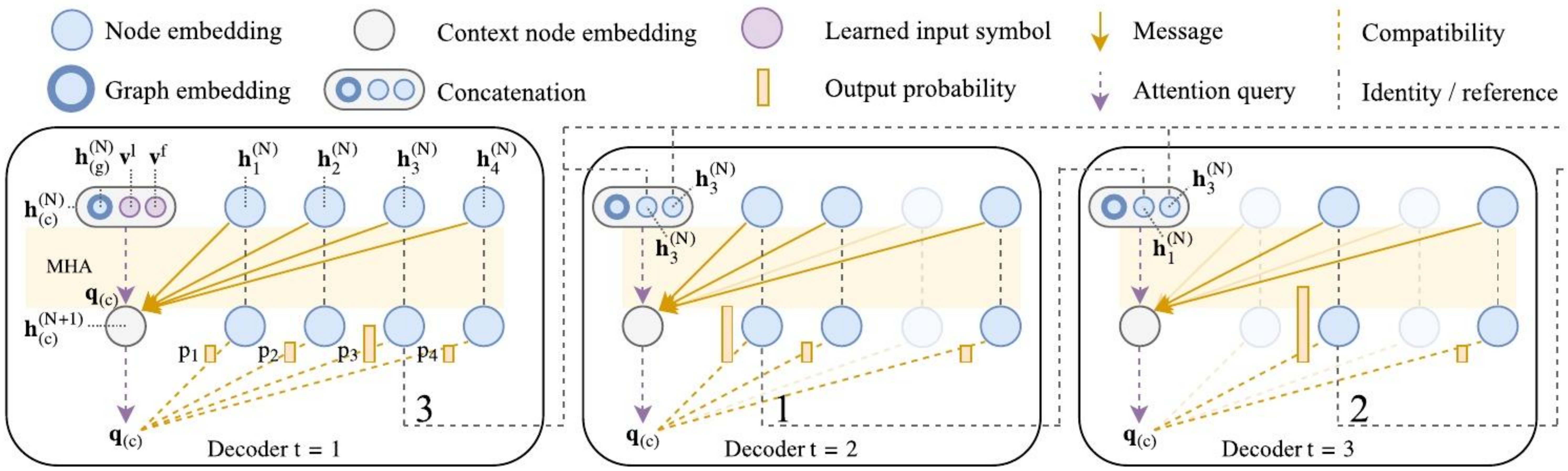


Figure: Decoder with output tour as (3,1,2,4)

Context Encoding:

$$\mathrm{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \mathbf{h}_{\pi_1}^{(N)}\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{v}^{\mathrm{l}}, \mathbf{v}^{\mathrm{f}}\right] & t = 1 \end{cases}$$

$$\mathrm{q}_{(c)} = W^Q \mathbf{h}_{(c)}, \mathbf{k}_i = W^K \mathbf{h}_i, \mathbf{v}_i = W^V \mathbf{h}_i$$

$$\mathrm{u}_{(c)j} = \begin{cases} C \cdot \tanh\left(\frac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_\mathrm{k}}}\right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise} \end{cases}$$

$$\mathrm{p}_i = p_\theta(\pi_t = i | s, \pi_{1:t-1}) = \frac{e^{u_{(c)i}}}{\sum_j e^{u_{(c)j}}}$$

where $u_{(c)j}$ are called compatibilities.

Using REINFORCE

From the probability distribution $p_\theta(\pi|s)$ obtained from the decoder for the problem instance s, we sample a policy π (permutation of nodes) to computed the loss defined as

$$\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[L(\pi)]$$

For REINFORCE we have the gradient of the loss as

$$\nabla\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[(L(\pi) - b(s))\nabla \text{log} p_\theta(\pi|s)]$$

A good baseline $b(s)$ reduces the gradient variance!
We use 2 baselines for the experimentation:

1. greedy rollout baseline: defined as the cost of a solution of a deterministic greedy rollout policy by the best model so far
2. critic: a function $\hat{V}s, w$ of state s, parameterized by w, which is learned using gradient ascent iteration similar to original REINFORCE iteration

More on Baseline

During the model training, the baseline is frozen for fixed number of steps every epoch.

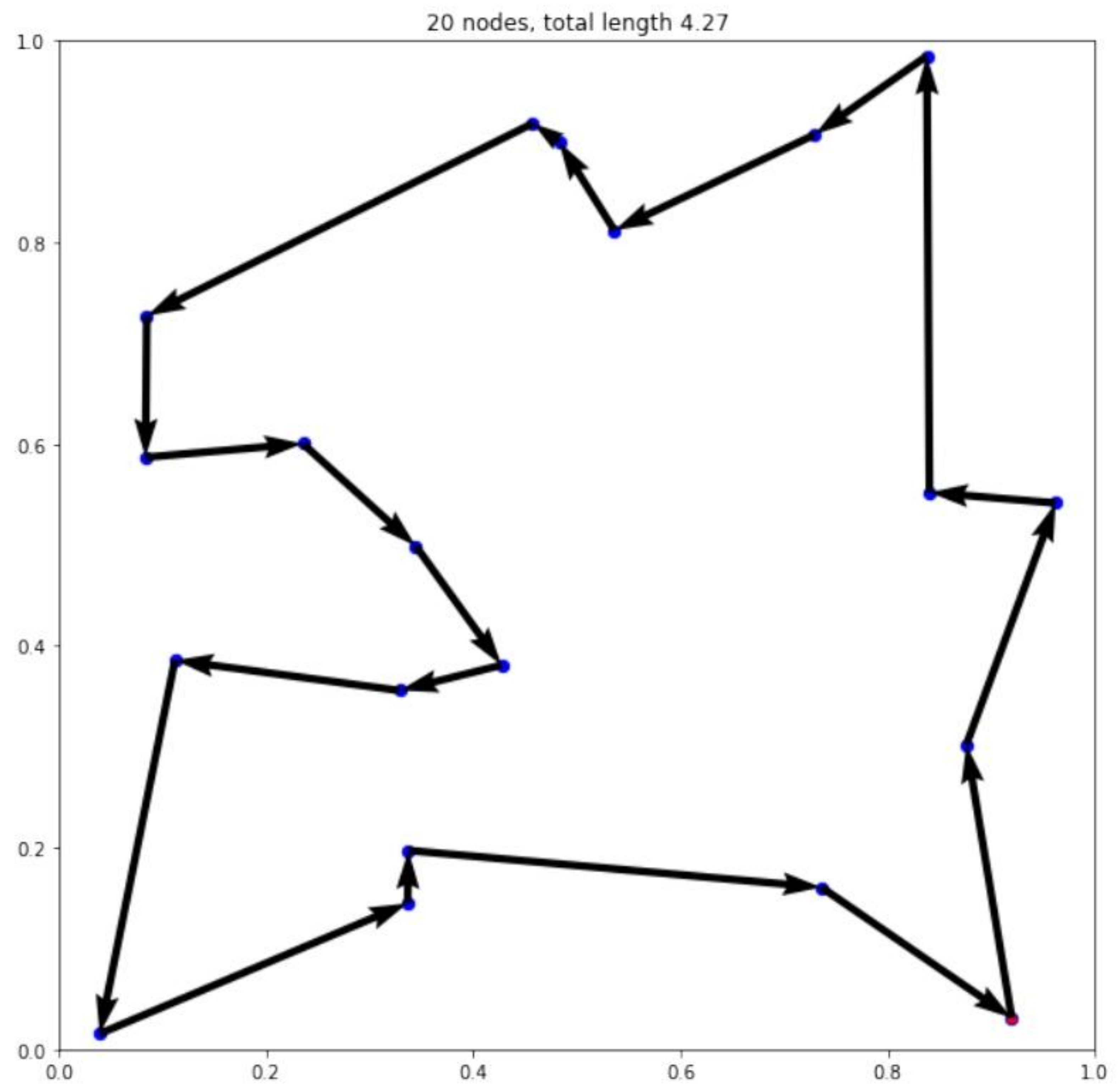
The parameters associated with baseline policy θ^{BL} is changed to policy parameters θ at the end of every epoch if current training policy is better compared to the baseline policy according to a paired t-test.

Overall Algorithm

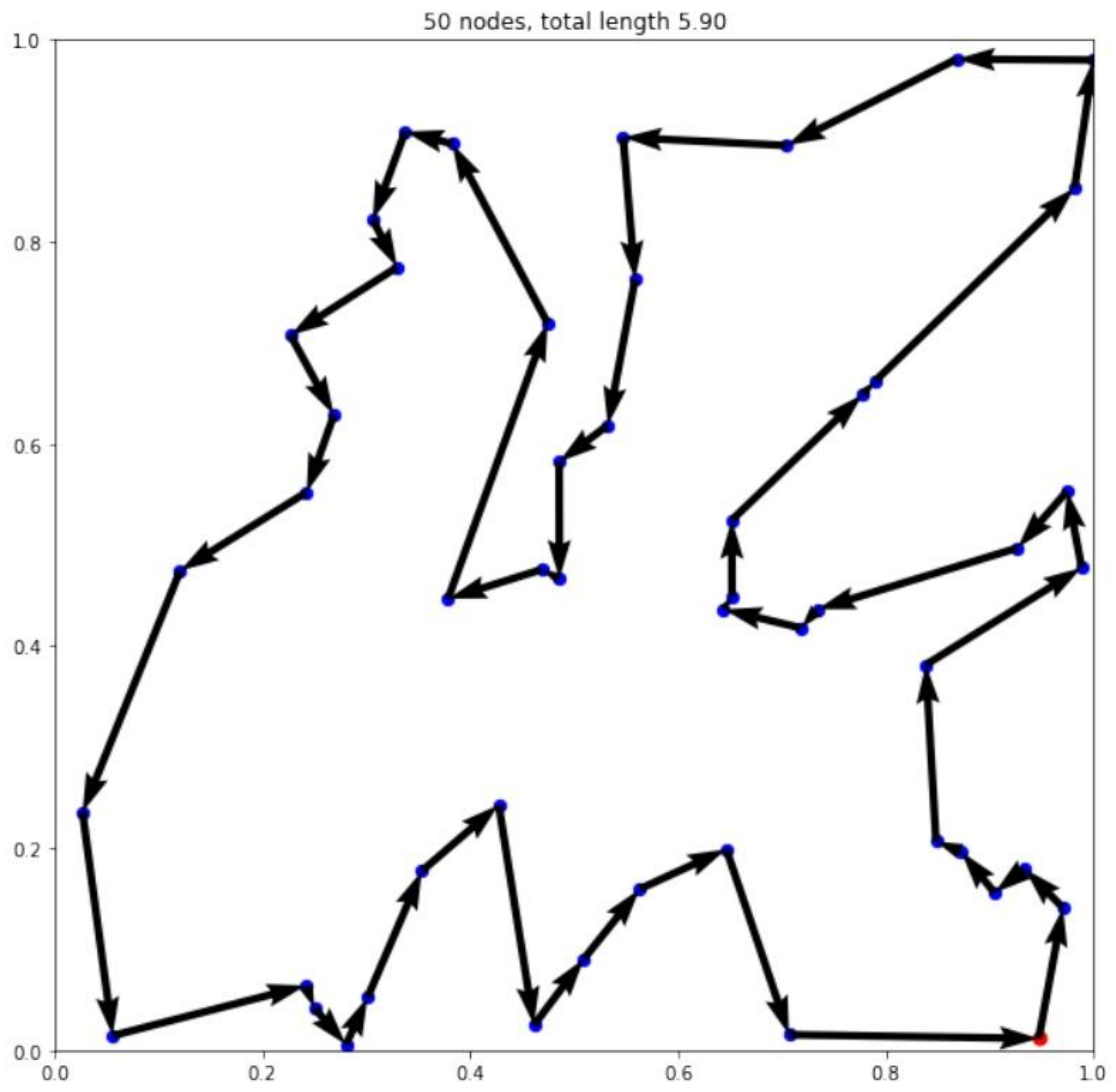
Algorithm REINFORCE with Rollout Baseline

```
**Input:** number of epochs $E$, steps per epoch $T$, batch size $B$, significance $\alpha$
Init $\theta,\ \theta^{\text{BL}} \leftarrow \theta$
for $epoch = 1, \dots, E$ do
    for $step = 1, \dots, T$ do
        $s_i \leftarrow \text{RandomInstance}()\ \forall i \in \{1, \dots, B\}$
        $\pi_i \leftarrow \text{SampleRollout}(s_i, p_\theta)\ \forall i \in \{1, \dots, B\}$
        $\pi_i^{\text{BL}} \leftarrow \text{GreedyRollout}(s_i, p_{\theta^{\text{BL}}})\ \forall i \in \{1, \dots, B\}$
        $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left( L(\pi_i) - L(\pi_i^{\text{BL}}) \right) \nabla_\theta \log p_\theta(\pi_i)$
        $\theta \leftarrow \text{Adam}(\theta, \nabla \mathcal{L})$
    end
    if $OneSidedPairedTTest(p_\theta, p_{\theta^{BL}}) < \alpha$ then
        $\theta^{\text{BL}} \leftarrow \theta$
    end
end
```

Experimental Results



(a) 20 Nodes



(b) 50 Nodes

Figure: Solving Traveling Salesman Problem

Experimental Results (contd.)

20 Nodes

50 Nodes

Experimental Results (contd.)

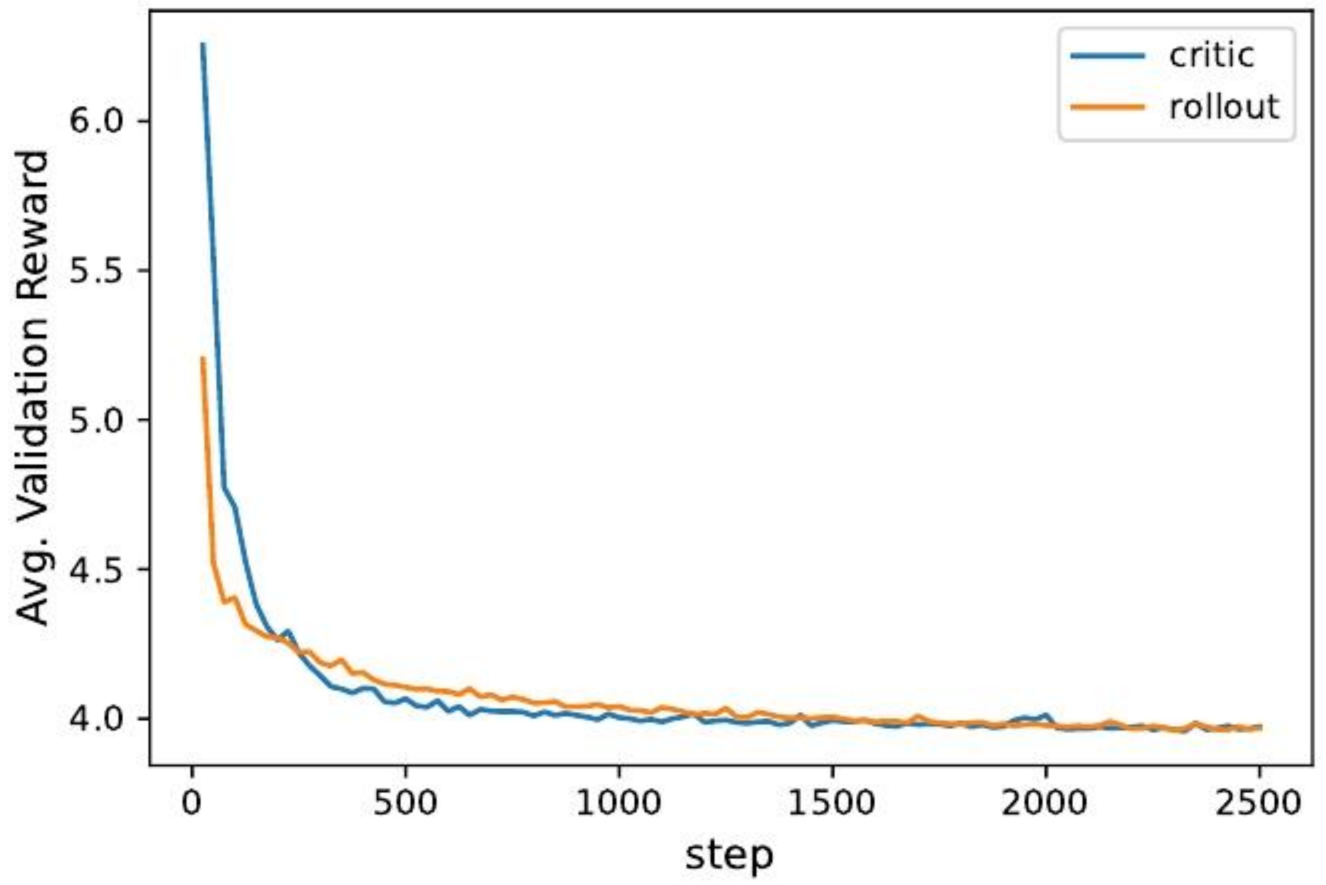
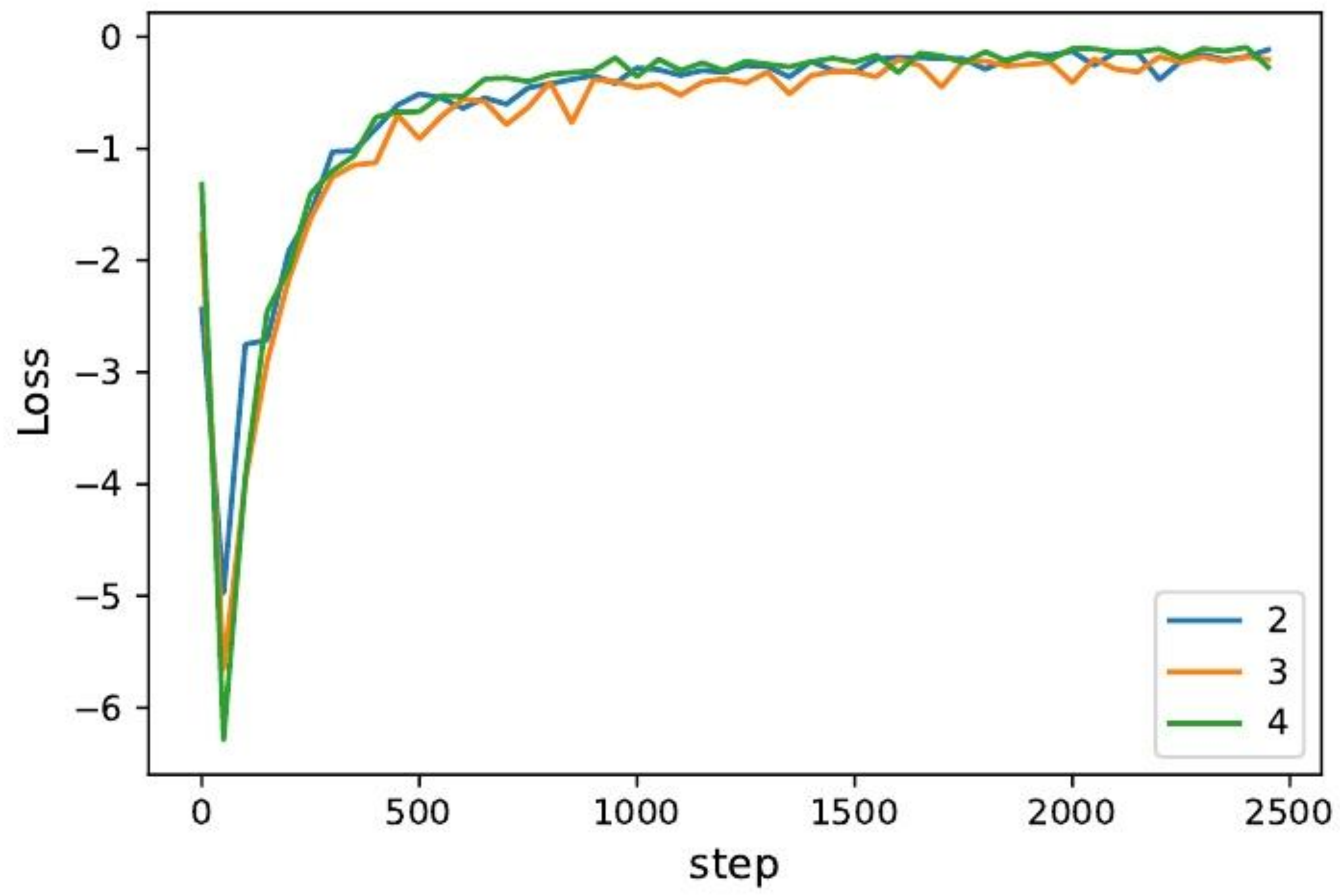
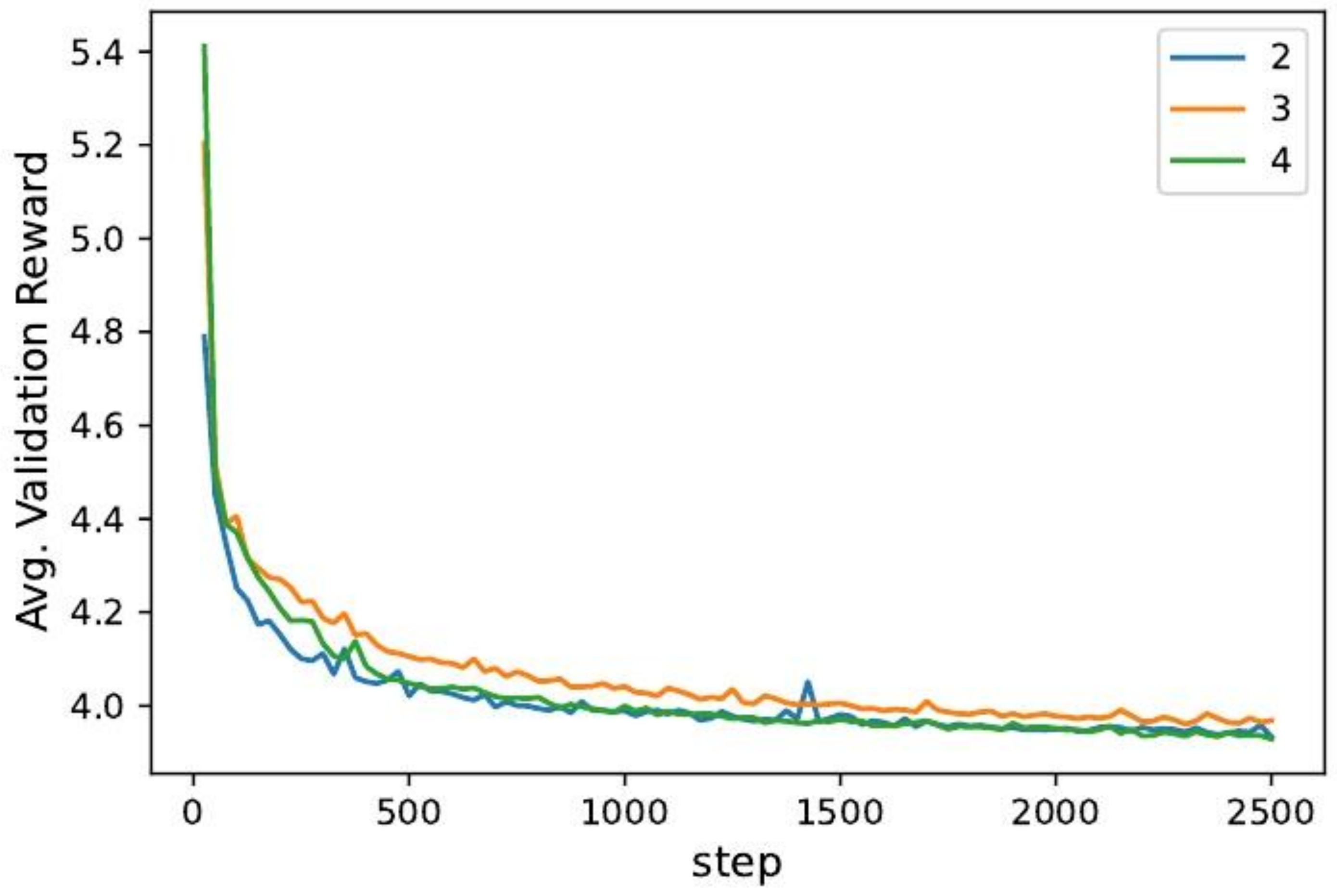


Figure: Different baseline: greedy rollout and critic



(a) Learning Loss



(b) Average Reward

Curriculum based learning?

Can we train the model on 20 nodes → 40 nodes → and so on → 100 nodes?

Does this improve performance?

The End